Chapter 3

Technical contributions

We aim to create Artificial Software Diversity for WebAssembly , by providing tools to make the process easier and feasible for developers and researchers. As far as we know, there is no software that provides Artificial Software Diversification for WebAssembly. Therefore, we need to enunciate the engineering foundation to implement the strategies in Section 2.2. Our implementations are part of the contributions of this thesis. Concretely, we provide two software artifacts that complement this work. Our approach generate WebAssembly program variants statically at compile time to provide randomization. Besides, it provides the tooling to generate MVE binaries for WebAssembly.

In this chapter we describe our technical contributions. In Section 3.1 we enunciate how the current state-of-the-art lead us to contribute with Software Diversification through LLVM. We follow by describing our two contributions and their main technical insights in Section 3.2 and Section 3.3. Besides, we describe a new transformation strategy as part of our contributions.

3.1 Artificial Software Diversity for WebAssembly

The work of Hilbig et al. [5] at 2021 influences our engineering decisions. According to their work LLVM-based compilers created the 70% of the WebAssembly binaries in the wild. Therefore, we decided to provide Artificial Sotfware Diversity for WebAssembly through LLVM. Other solutions would have been to diversify at the source code level, or at the WebAssembly binary level. However, the former would limit the applicability of our work. The latter, will be addressed in future works (Section 6.2).

LLVM is composed by three main components [?] . First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation passes improve the LLVM IR. Third and final, the backend component is in charge of generating the target machine code. Notice that, the LLVM architecture is highly

scalable. The machine code translation of LLVM IR might have any number of custom intermediate passes. In Figure 3.1 we show the generic workflow followed in our contributions. In the context of our work the LLVM architecture is instantiated over all LLVM frontends (1), it adds a Diversifier as a LLVM IR pass (2) and uses a custom Wasm backend (3). The dashed squares in Figure 3.1 wrap the components for which we contribute.

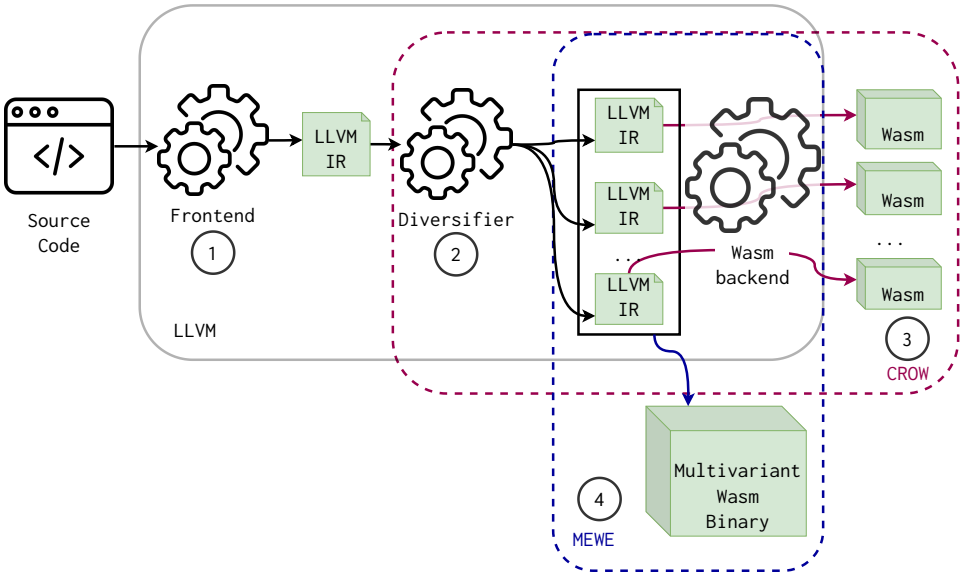

Figure 3.1: Generic workflow to create WebAssembly program variants.

The generic workflow in Figure 3.1 starts by receiving the source code from the program to be compiled to WebAssembly . Then, an LLVM frontend transforms this source code into LLVM IR representation. The resulting LLVM IR is the input for the Diversifier (1). The diversifier generates LLVM IR variants from of the output of the frontend (2). These variants are inputs for our customized Wasm backend. In our case the LLVM backend is always for WebAssembly and, it finalizes the creation of the variants (3).

Our first technical contribution, CROW [8], includes the implementation of the diversifier for LLVM and the customized WebAssembly backend. CROW is able to create several WebAssembly program variants out of a source code program. In Section 3.2 we dissect CROW into more details. In addition, an orthogonal contribution comes from the generation of LLVM IR variants at Step (2). Our second contribution, MEWE [7], merges and creates multivariant binaries to provide MVE for WebAssembly (4). In Section 3.3 we describe MEWE in details.

3.2 CROW: Code Randomization Of WebAssembly

This section describes CROW [8], our first contribution. Following the workflow in Figure 3.1, CROW is a tool tailored to create semantically equivalent WebAssembly variants out of LLVM IR passed through the LLVM frontend to be compiled as WebAssembly code. In Figure 3.2, we describe the architecture of CROW to create program variants. The figure highlights the main two components of the Diversifier, *exploration* and *combining*. The workflow starts by passing the input LLVM IR to perform the *exploration*. During the *exploration* process, at the instruction level for each function in the input LLVM IR, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM bitcode variants. Then, the corresponding backend compiles the LLVM IR variants into WebAssembly binaries.

In the following, we describe our engineering decisions. All our implementation choices are based on one premise: each implementation decision should increase the number of WebAssembly variants that CROW creates.

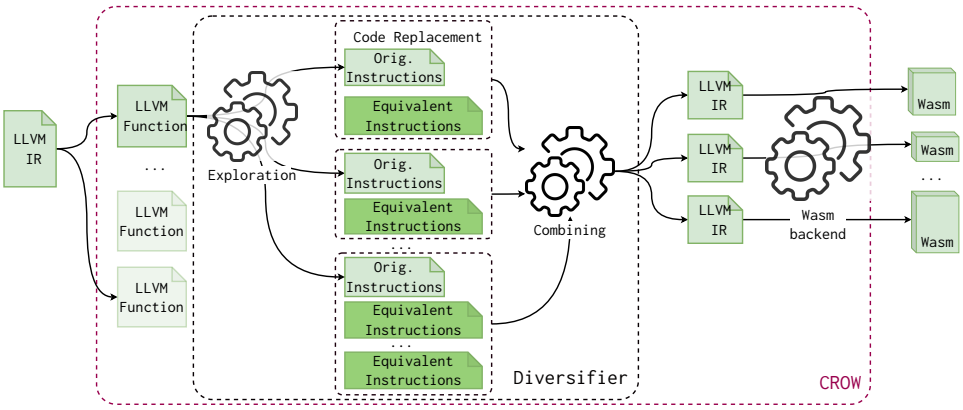

Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM bitcodes to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

Exploration

The key component of CROW if its code replacements generation strategy. The diversifier implemented in CROW is based on the work of Jacob et al. [52]. Their work uses code superoptimization to generate software diversification for x86 with an approach called superdiversification. Jacob and colleagues cornerstone, code superoptimization, focuses on *searching* for a new program which is faster or

smaller than the original code, while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [67] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. The main difference between the superoptimization process and a superdiversifier it that the latter keeps all intermediate search results despite their performance.

We use the seminal work of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on hand-written rules. Besides, this technique is fully automatic. Second, there is a battle tested superoptimizer for LLVM, Souper [30]. This latter, makes feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. The searching algorithm of Souper is based on inferring the smallest Data Flow Graph for all integer returning instructions in the input LLVM IR. For a given integer returning instruction, it exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original. The search halts when a semantically equivalent expression is found. Souper synthesizes the replacements in increasing size, thus, the first found equivalent transformation is the best and is the result of the searching. Instead of stopping the process as soon the first equivalent transformation is found, we remove it, keeping more equivalent replacements during the searching.

Notice that the searching space exponentially increases with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions that have no correspondence in the WebAssembly backend. This decision reduces the searching space. For example, creating expression having the `freeze` LLVM instructions will increase the searching space for an instruction without a Wasm suitable opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more variants. For example, Souper avoids to construct redundant expressions such as commutative operations. We disable strategies like this for the sake of more statically different programs.

As we discussed in Section 2.2, the equivalence checking is an important part of any program transformation process. In the case of CROW, we quarantee the equivalence property for program variants through Souper as well. Souper uses Z3 [?] , an SMT solver, to check for programs equivalence. Besides, as a sanity check, it uses KLEE and Alive to double-check that the generated LLVM IR binary out of the code replacement is valid.

Constant inferring

By extending Souper as a superdiversifier, we contribute with a new mutation strategy, *constant inferring* (in addition to the before mentioned strategies in Section 2.2). The main component of Souper infers pieces of code as a single constant assignment particularly for boolean valued variables that are used to control branches. If a program branching is removed due to a constant inferring, the generated program is considerably different to the original program, statically and dynamically.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop which stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) % 1000000
            ↪  != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n", current);
8       return 0 ;
9   }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. Souper can deal with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code, and is removed in the final compilation. It is clear that the new program in Listing 3.2 is remarkably smaller and faster than the original code. Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also notice that the two programs in the example follow the definition of *functional equivalence* discussed in Section 2.2.

Combining replacements

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of

performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformation only to the instruction for which it was found in the first place. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

Removing latter optimizations for LLVM

During the implementation of CROW we have the premise of removing all builtin optimizations in the LLVM backend that could reverse Wasm variants. Therefore, in addition to the extension of Souper, we modify the LLVM compiler and the WebAssembly backend. We disable all optimizations in the WebAssembly backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

3.2.1 CROW instantiation

Let us illustrate how CROW works with the simple example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

Listing 3.3: C function that calculates the quantity $2x + x$

```
int f(int x) {
    return 2 * x + x;
}
```

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program.

Listing 3.4: LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.

```
define i32 @f(i32) {
        code 2
        code 1
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

ret i32 %3
}

define i32 @main() {
%1 = tail call i32 @f(i32
     10)
ret i32 %1
}
```

```
Replacement candidates for
      code_1

%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2

%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.5: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

Listing 3.6: Wasm program variants generated from program Listing 3.3.

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 2
   i32.mul
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 1
   i32.shl
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   local.get 0
   i32.add
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 3
   i32.mul
```

3.3 MEWE: Multi-variant Execution for WEbAssembly

This section describes MEWE [7], our second contribution. The core idea of MEWE is to synthesize diversified function variants, using CROW, providing execution-path randomization in an MVE. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step (2) in Figure 3.1 shows. CROW generates each one of these variants with fine-grained diversification at instruction level, applying the majority of the strategies discussed in Section 2.2 and *constant inferring*. Besides, MEWE inlines function variants when appropriate, also resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE((4)) from the diagram in Figure 3.1. MEWE takes the LLVM IR variants generated by our diversifier and merges them into a Wasm multivariant. In the figure, we highlight the two components of MEWE. In Step (I), we merge the LLVM IR variants created by CROW and we create a LLVM multivariant binary. In Step (II), we use a special component, called a "Mixer", which augments the binary with a random generator, which is required for performing the execution-path randomization. Also at this stage, the multivariant binary is fixed with the entrypoint of the original binary. The final output of Step (III) is a standalone multivariant WebAssembly binary that can be directly deployed. The source code of MEWE can be found at **TODO** .

Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function that is in charge of choosing a variant

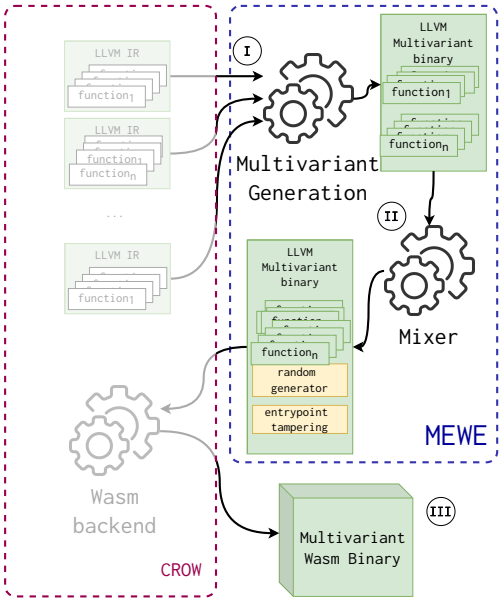

Figure 3.3: Overview of MEWE. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary and then generates a LLVM multivariant binary composed of all the function variants. Also, it includes the dispatcher functions in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and a tampering of the original application entrypoint, in order to produce a WebAssembly multivariant binary ready to be deployed.

at random, every time the original function is invoked during the execution. With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[69], where the nodes are typed. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In Figure 3.4, we show the original static call graph for and original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original program includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for

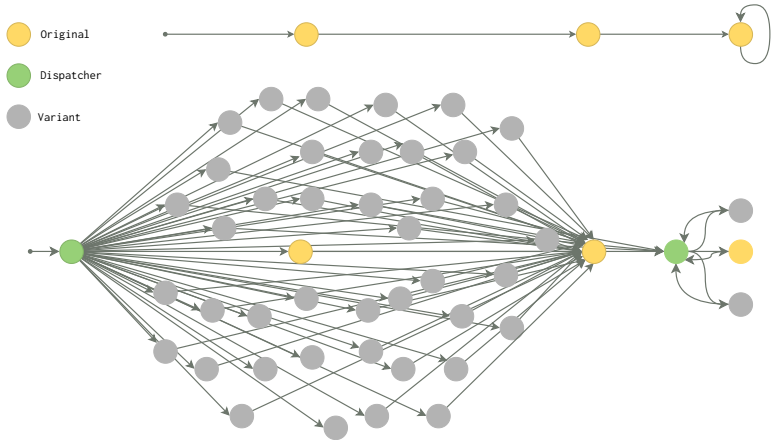

Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value that is then used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [2]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [4]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

Mixer

The Mixer has four specific objectives: tamper the entrypoint of the application, link the LLVM multivariant binary, inject a random generator and merge all these components into a multivariant WebAssembly binary. We use the Rustc compiler[1] to orchestrate the mixing. For the random generator, we rely on

[1] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.7: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

WASI's specification [9] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed.

The MEWE mixer creates a new entrypoint for the binary called *entrypoint tampering*. It simply wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

Conclusions

This chapter discusses the technical details for the artifacts implemented for our two contributions. We describe how CROW generates program variants. We introduce a new mutation strategy that is a consequence of retargeting a superoptimizer for LLVM as a superdiversifier. Besides, we dissect MEWE and how it creates an MVE system. In Chapter 4 we discuss the methodology we follow to evaluate the impact of CROW and MEWE.